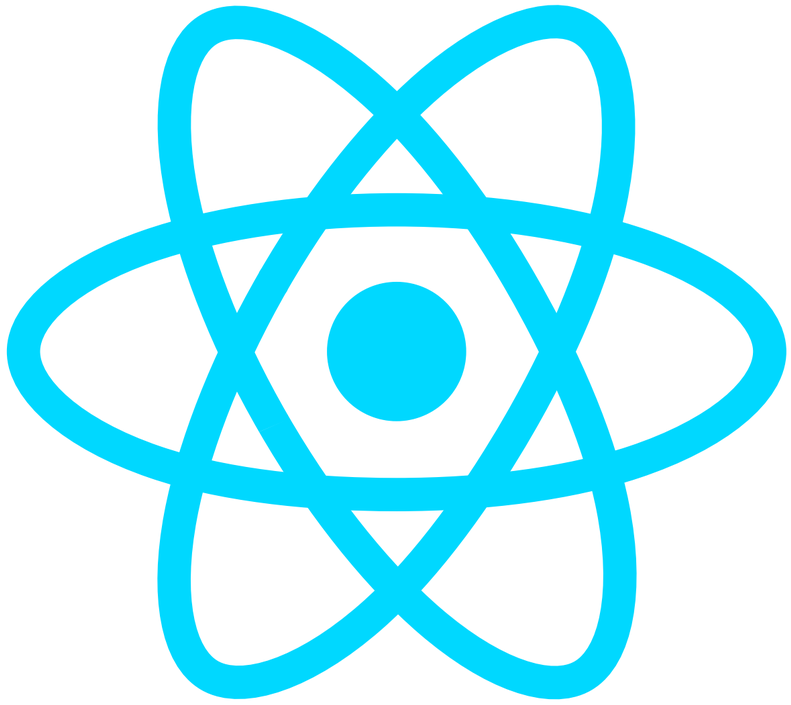


React

RESUMO DE AULAS

SUMÁRIO:

INTRODUÇÃO:

Para poder ter total compreensão do react e os assuntos abordados é preciso ter conhecimentos prévios em:

- html;
- css;
- javaScript;

INTRODUÇÃO:

O que é:

React JS é uma biblioteca JavaScript para a criação de interfaces de usuário ou UI (user interface).

Criado em 2011 pelo time do Facebook, o React surgiu com o objetivo de otimizar a atualização e a sincronização de atividades simultâneas. Com o React é simplificada a conexão entre HTML, CSS e JavaScript e todos os componentes de uma página.

Hoje, o React é uma das mais prestigiadas bibliotecas de JavaScript do mercado.

INTRODUÇÃO:

Como funciona:

O React é uma biblioteca front-end e tem como um de seus objetivos facilitar a conexão entre diferentes partes de uma página, portanto seu funcionamento acontece através do que chamamos de componentes.

Em outras palavras, podemos imaginar que o React divide uma tela em diversos componentes para, então, trabalhar sobre eles de maneira individual.

Os componentes são utilizados para reaproveitamento de código e padronização de interface.

Isso torna o React uma tecnologia muito flexível para a solução de problemas e para a construção de interfaces reutilizáveis, uma vez que cada um destes componentes pode ser manipulado de maneira distinta.

INTRODUÇÃO:

Principais comandos:

Para poder iniciar um projeto em react existem diversos comandos, porém o mais utilizado é o:

- npx create-react-app *nome do projeto*

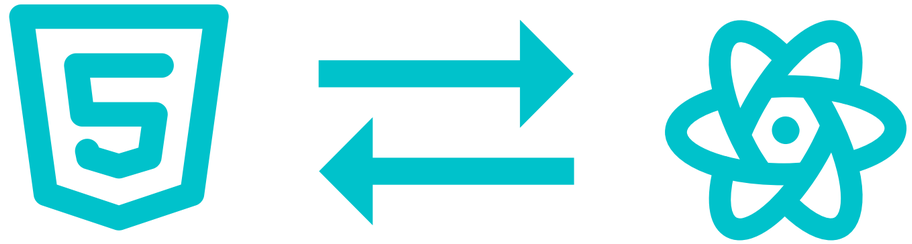
Para poder instalar as dependecias do projeto, usamos o:

- npm install

Para poder iniciar o projeto no localhost usamos o:

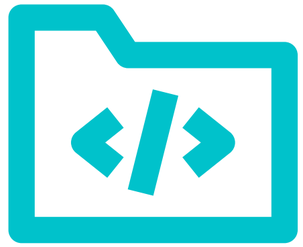
- npm start

JSX:



- O JSX é como um **HTML**, porém dentro do código JavaScript;
- É a principal maneira de escrever HTML com o React;
- É possível **interpolar variáveis**, inserindo ela entre **{}**;
- É possível também **executar funções** em JSX;
- Insirir valores em atributos de tags também é válido em JSX.

JSX:



Criado pela equipe de desenvolvimento do React, o JSX é uma **forma de criar elementos para serem utilizados como templates** de aplicações React. Basicamente, os elementos criados com o JSX são bem **similares com código HTML** e fornecem aos desenvolvedores uma forma mais simples e intuitiva de criar os componentes de uma aplicação.

Porém, apesar de muito similar ao HTML, **o JSX não é interpretado pelo navegador**. Por este motivo, deve-se utilizar um “transpilador” para essa conversão. Atualmente, o mais conhecido deles é o Babel.

JSX:

No exemplo podemos ver a construção de uma nav em JSX, ao lado temos o mesmo elemento sendo construido com JavaScript puro.

JSX:

```jsx
const nav_jsx = (
    <nav>
      <ul>
        <li><a href="#">Início</a></li>
      </ul>
    </nav>
)
```

JS:

```js
var nav_js = React.createElement(
    "nav",
    { className: "links" },
    React.createElement(
        "ul",
        null,
        React.createElement(
            "li",
            null,
            React.createElement(
            "a",
            { href: "#" },
            "Inicio"
            )
        ),
    )
);
```

COMPONENTS:

- Permite **dividir a aplicação em partes**;
- Os componentes **renderizam JSX**, assim como App.js (que é um componente);
- Precisamos **criar um arquivo** de componente;
- E **importá-lo** onde precisamos utilizar;
- Normalmente ficam em uma **pasta chamada components**;

COMPONENTS:

Componentes permitem você **dividir a UI em partes independentes, reutilizáveis e separando tudo isoladamente.** Conceitualmente, componentes **são como funções JavaScript**. Eles aceitam entradas arbitrárias (chamadas “props”) e **retornam elementos React** que descrevem o que deve aparecer na tela.

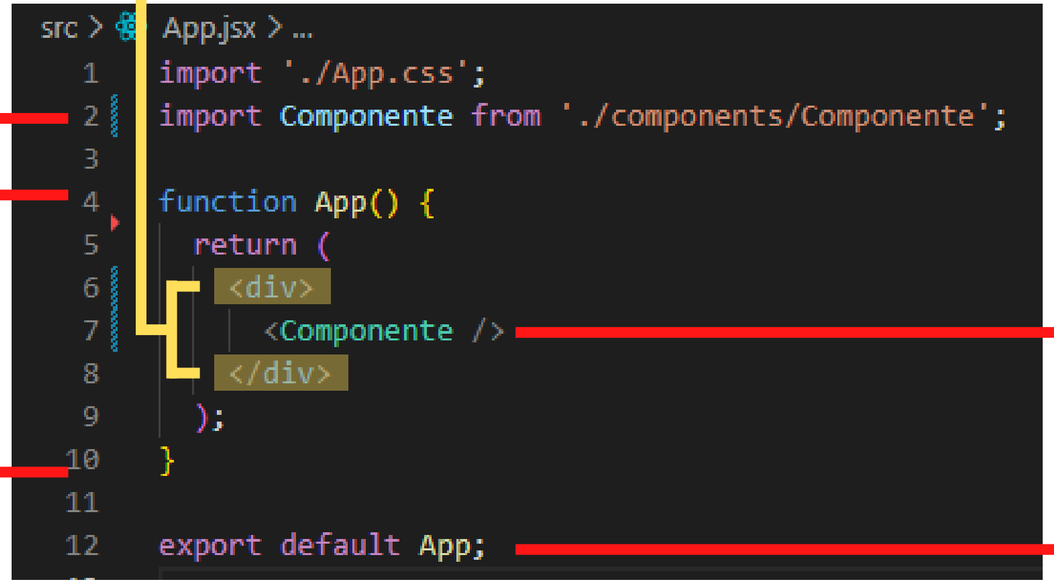
COMPONENTS:

exemplo:

Abaixo é possível ver o que seria a construção de um componente, no caso o componente "App" que normalmente costuma ser o componente principal do projeto que engloba outros componentes:

importação do componente "Componente".

A função App corresponde ao que seria o componente principal do projeto que engloba outros componentes.



Observe que o componente está envolvido por uma DIV, isso é muito importante pois o retorno da função sempre tem que ser envolvido por apenas uma única tag, uma função não pode retornar mais de uma tag.

Aqui temos o que seria a "instanciação" do componente importado no início do código.

Exportação da função (componente) criada, no caso o "App".

COMPONENTS:

Dentro do componente mostrado temos uma constrição simples de um parágrafo escrito "Ola Mundo!".

Podemos ver que a sua construção é muito semelhante ao do App, uma vez que ambos são componentes, a diferença é que um engloba o outro.

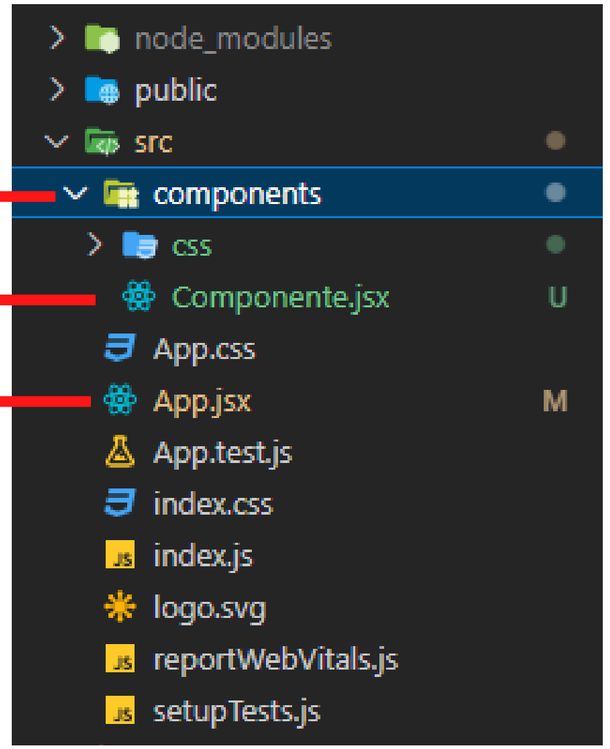
```jsx
function Componente(){
    return(
        <div>
            <p>
                Hello World!
            </p>
        </div>
    );
}

export default Componente;
```

Pasta criada para guardar os componentes.

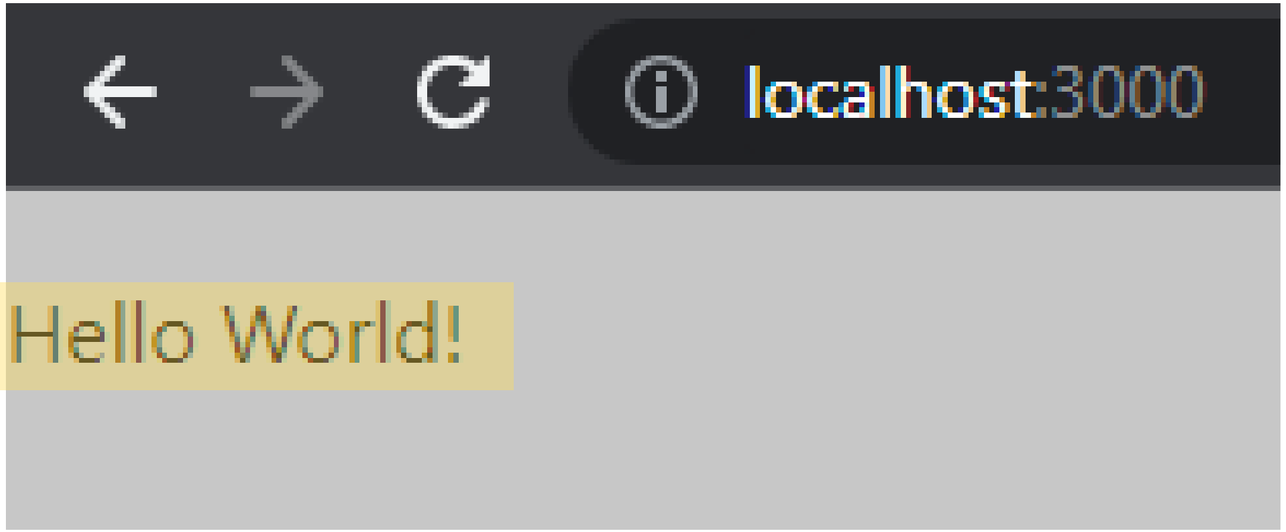
Componente criado e utilizado no App.

Componente pai que importa os demais componentes existentes.



COMPONENTS:

Aqui temos o resultado do nosso projeto aparecendo no navegador após realizar as etapas dos slides anteriores. Uma simples demonstração de um parágrafo com "Hello world!".



PROPS:

- As props são valores passados para os componentes;
- As Props **podem deixar o componente dinâmico**, ou seja, mudando a execução por causa do seu valor;
- O valor é passado como um **atributo** na chamada do componente;
- Precisa ser resgatado dentro de uma propriedade/argumento chamada props na função de definição do componente;
- As props são **somente de leitura**.

- Podemos **definir tipos para as props**, realizando uma espécie de validação;
- A definição é feita em um objeto chamado **propTypes** no próprio componente;
- Para poder definir um valor padrão para as Props podemos utilizar do objeto **defaultProps**;

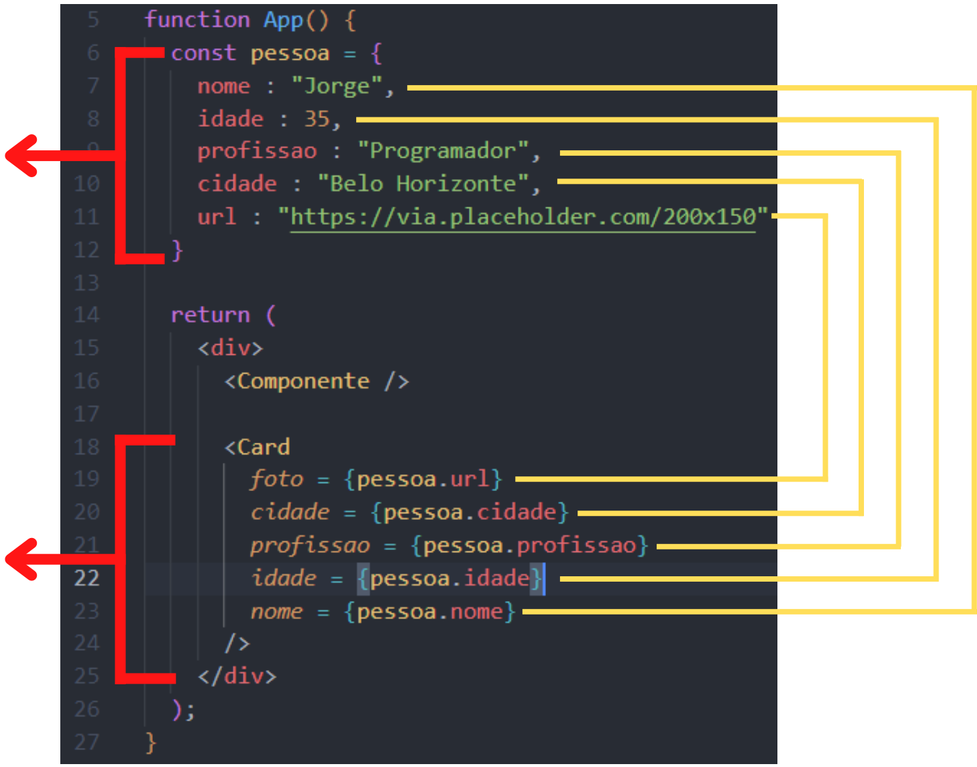
João Vitor Montanari

PROPS:

O props são "parâmetros" passados para um componente que serão mostrados na aplicação. E para poder ilustrar esse funcionamento criamos o componente "card".

Aqui temos a construção de um objeto chamado "pessoa" com diferentes campos que descrevem o perfil de alguém que queremos mostrar no front-end.

"Instanciamos" o componente Card passando para ele os campos que estão presentes no objeto pessoa, referenciando o nome, idade, profissao, cidade e foto.



PROPS:

Olhando agora para dentro do componente Card podemos ver como ele lida com as informações passadas para ele no componente principal da aplicação (App). Aqui essas informações são formatadas para que apareçam no navegador da forma desejada.

```
1  import Styles from './css/Card.module.css';
2
3  function Card(props){
4      return(
5          <div className={Styles.container}>
6              <h1>{props.nome}</h1>
7              <ul>
8                  <li>Idade: {props.idade}</li>
9                  <li>Cidade: {props.cidade}</li>
10                 <li>Profissão: {props.profissao}</li>
11             </ul>
12             <img src={props.foto} alt={props.nome} />
13         </div>
14     );
15 }
16
17 export default Card;
```

Aqui estamos resgatando os dados passados por meio da props, dessa forma podemos alinhar as informações do componente App com o componente Card.

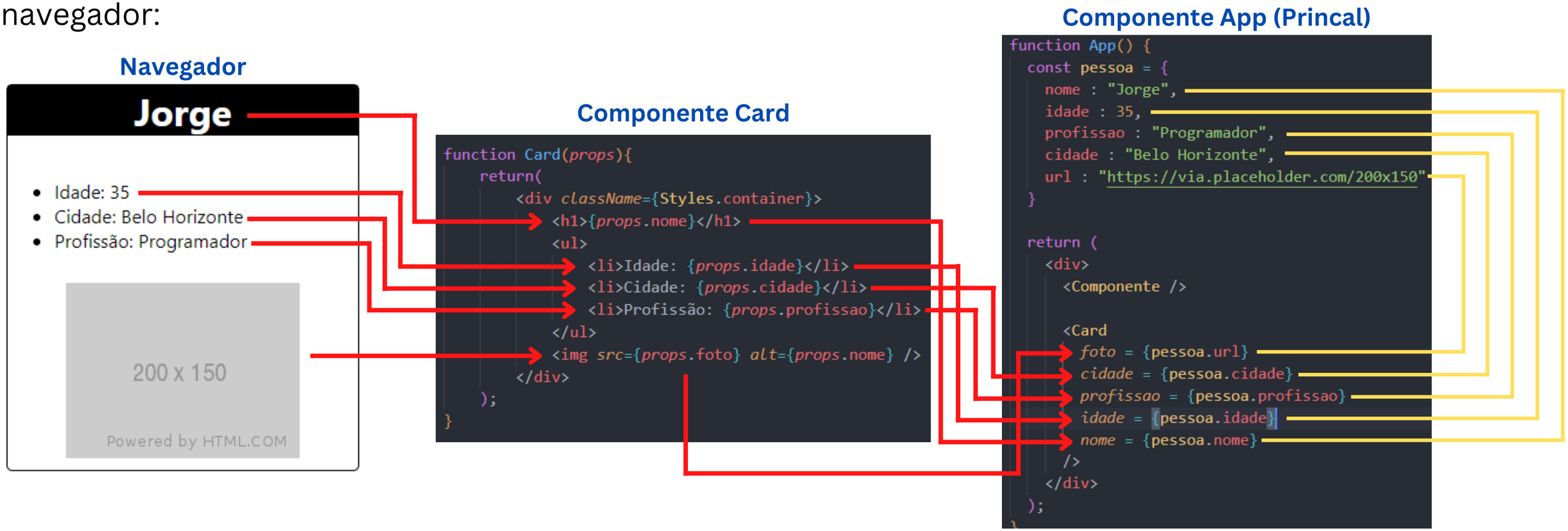
Repare que toda vez que queremos utilizar um dos dados passados para o componente devemos fazer referência ao props e envolve-la com chaves {}, dessa forma é possível identifica-la, fazemos isso com:

- Nome;
- Idade;
- Cidade;
- Profissão;
- Foto.

As partes marcadas em azul dentro do componente Card fazem referência a estilização do mesmo. A estilização (CSS Modules) se trata do proximo tema que vai ser abordado nos slides, então se preocupe caso haja dúvidas relacionadas ao tema.

PROPS:

Observe agora o caminho e o resultado da aplicação desse componente a partir do projeto até o navegador:



OBS: Lembrando que o resultado só se apresentou dessa forma uma vez que utilizamos de estilização por meio do CSS Modules. Caso você não tenha utilizado de tal recurso você verá os resultados da props da mesma forma, porém sem um contexto de estilo.

PROPS:

Continuando com props vamos passar para um assunto mais profundo relacionado ao tema, falando um pouco sobre **propTypes** e **defaultProps**. Ambos são importantes para que possamos lidar com as adversidades relacionadas ao props que chega ao componente.

O **propTypes** especificamente tem a função de definir a tipagem esperada da props que está chegando no componente. Ou seja, nele podemos dizer o que aquela informação tem que ser, seja um número, uma string, etc.

```
import PropTypes from 'prop-types';
```

Importação da propriedade prop-types da biblioteca react.

```
Card.propTypes = {
    nome : PropTypes.string.isRequired,
    idade : PropTypes.number,
    cidade : PropTypes.string.isRequired,
    profissao : PropTypes.string.isRequired
}
```

Nesses dois campos definimos o tipo da props nome e idade, colocando um como string e outro como número. Logo o esperado como entrada são valores que atendam a esses aspectos.

```
Warning: react-jsx-dev-runtime.development.js:87
Failed
prop type: Invalid prop `idade` of type `string`
supplied to `Card`, expected `number`.
    at Card (http://localhost:3000/static/js/bundle.j
s:129:23)
    at App
```

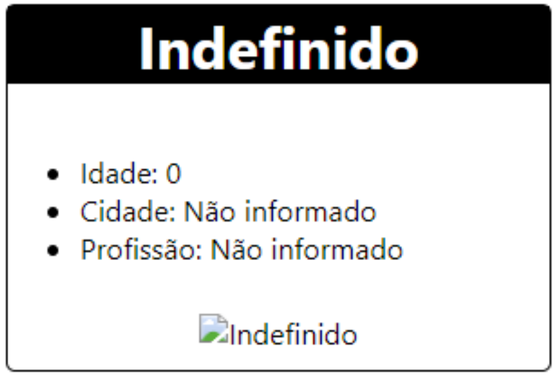
Caso o props não atenda ao requerimento do PropTypes será possível visualizar um aviso de prop invalido no console do navegador.

PROPS:

Outra ferramenta que podemos utilizar é o defaultProps que tem a funcionalidade de deixar o props com um valor padrão caso o mesmo não seja definido na "instanciação" do componente.

No exemplo definimos os valores default do nome, idade, cidade e profissão. Assim quando esse componente "Card" for criado sem esses parâmetros veremos o mesmo sendo preenchido por esse valores predefinidos como padrão.

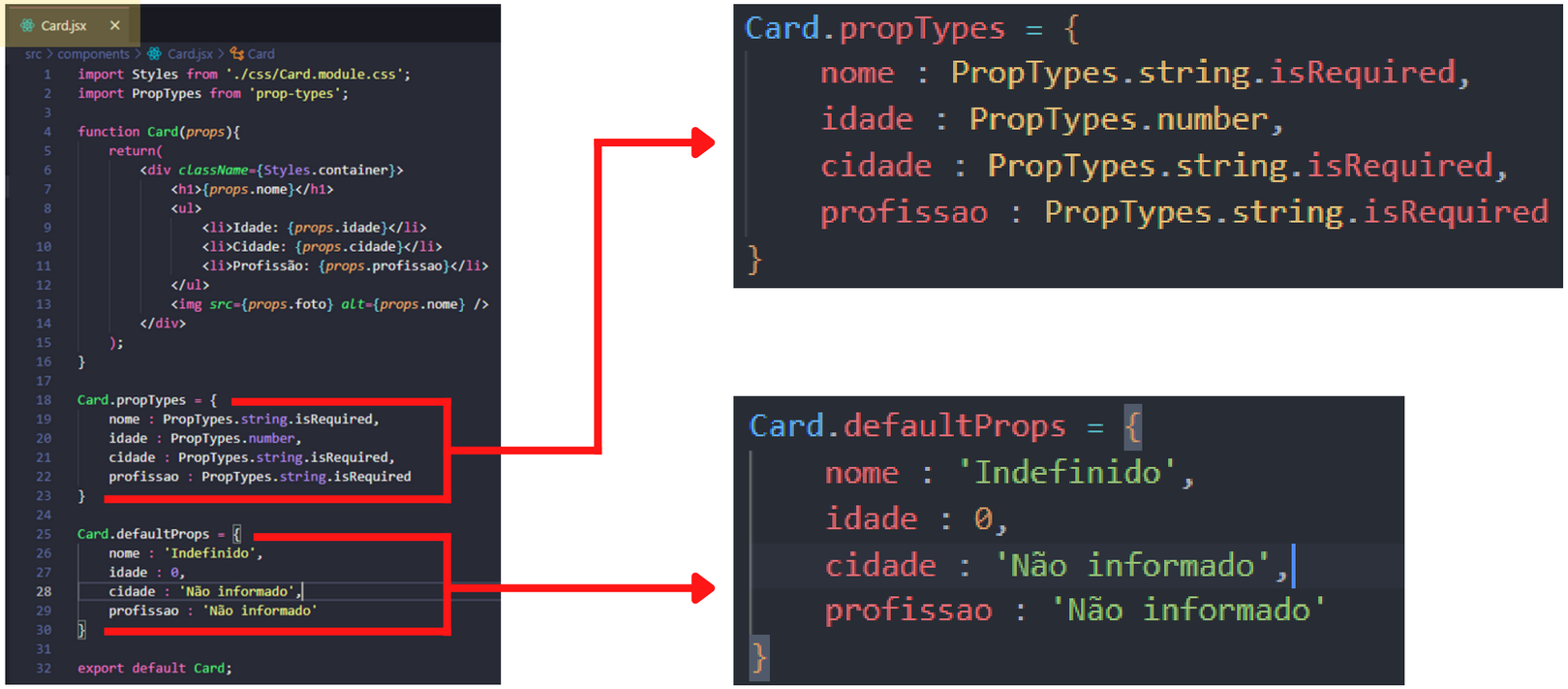
```
Card.defaultProps = {
    nome : 'Indefinido',
    idade : 0,
    cidade : 'Não informado',
    profissao : 'Não informado'
}
```



Aqui podemos ver o resultado no navegador, caso o componente seja chamado sem props definidos.

PROPS:

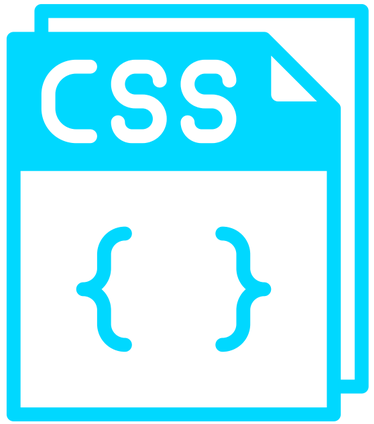
Importante pontuar que todas essas definições são feitas dentro do componente! Assim como no exemplo, na onde dentro do Card definimos os valores default das Props e os seus tipos.



OBS: Não esqueça de importar o PropType dentro do componente para poder usa-lo.

CSS MODULES:

- O CSS pode ser adicionado de forma global na aplicação, no react ele pode ser definido desta forma por meio do arquivo **index.css**;
- O CSS Module é a **estilização por nível de componente**;
- Tem a utilidade de estilizar o nosso projeto **por partes**;



CSS MODULES:

O CSS Modules se trata da estilização por **nível de componente**, permitindo que o desenvolvedor possa focar no estilo de **apenas uma parte do programa**, isso acaba gerando uma maior **facilidade** no momento de criar estilizações, garantindo que o CSS **não fique complexo** e cheio de classes centralizadas em apenas um arquivo. Assim evitando confusões e conflitos.

CSS MODULES:

Para poder estilizar o componente Card que utilizamos como exemplo mais cedo nos slides criamos o "Card.module.css". Para poder utiliza-lo devemos importa-lo no componente com seu caminho.

App.jsx | Card.jsx | Card.module.css

src > components > Card.jsx > Card

```jsx
import Styles from './css/Card.module.css';

import PropTypes from 'prop-types';

function Card(props){
    return(
        <div className={Styles.container}>
            <h1>{props.nome}</h1>
            <ul>
                <li>Idade: {props.idade}</li>
                <li>Cidade: {props.cidade}</li>
                <li>Profissão: {props.profissao}</li>
            </ul>
            <img src={props.foto} alt={props.nome} />
        </div>
    );
}
```

Importação do arquivo de estilização CSS Module com a nomeação "Styles".

referenciando a classe "container" para a estilização.
Veja que não se deve utilizar "class" como é feito normalmente no HTML, no lugar disso utilizamos o "className".

CSS MODULES:

No exemplo abaixo já podemos ver como é desenvolvido o arquivo CSS Modules, claramente não existem mudanças para o CSS tradicional que já é utilizado nos projetos normais que envolvem html. Ao lado temos o o que seria o resultado Card que vai se apresentar no navegador.

```css
.container{
    width: 300px;
    display: grid;
    margin: 40px auto;
    border-radius: 5px;
    border: 1px solid #000;
}

.container h1{
    color: #fff;
    margin-top: 0px;
    text-align: center;
    background-color: #000;
}

.container img{
    margin: 10px auto;
}
```

Observe que com poucas linhas de código é possível estilizar um Card inteiro sem complicações.

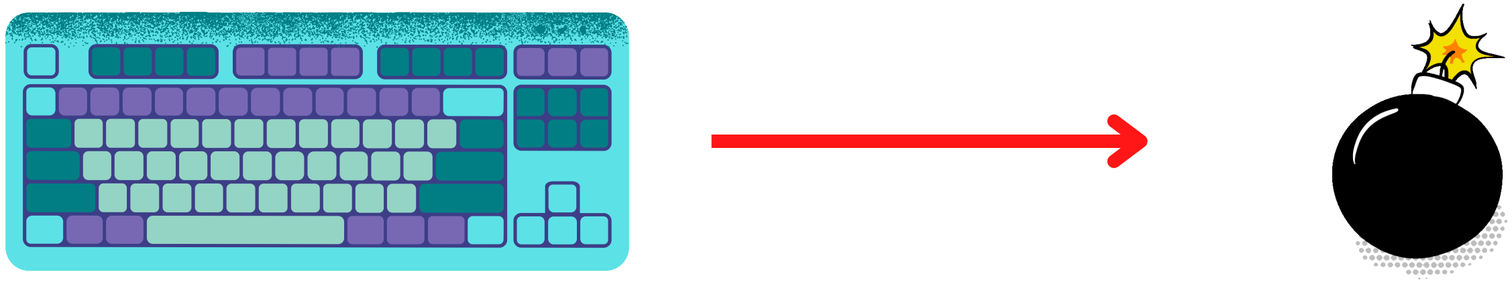
Esse se trata do resultado que vai ser observado no navegador. Veja que a organização se dá em grande parte por conta do CSS Modules.

EVENTOS:

- Os eventos do React são os mesmo do DOM;
- Existem eventos que respondem a um **click**;
- O evento é **atrelado a uma tag** que irá executá-lo;
- Geralmente um método é **atribuído** ao evento;
- O método **deve ser criado no componente**;

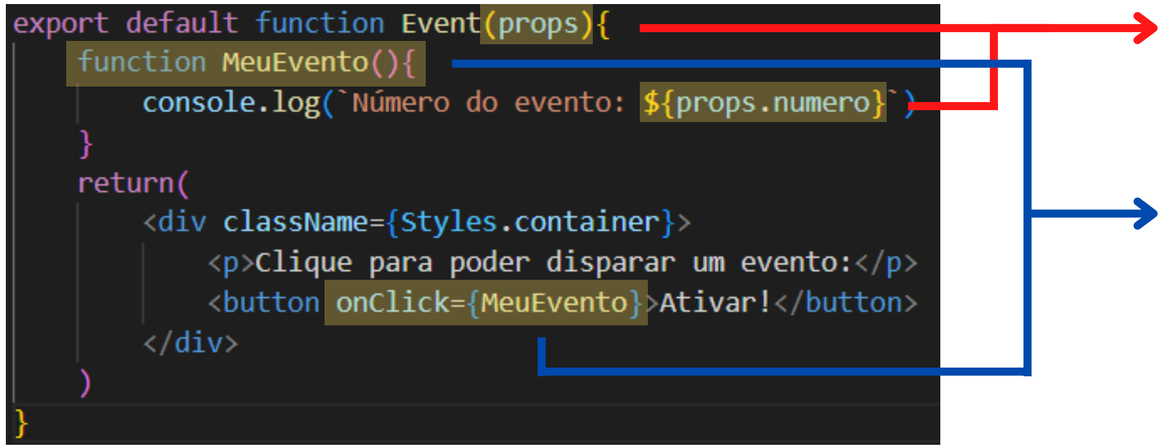
EVENTOS:

Os Eventos se tratam de **chamadas disparadas** sempre que alguma coisa específica acontece dentro do seu código. Quando uma pessoa usuária pressiona uma tecla em um teclado, por exemplo, o programa em execução recebe um evento “KeyDown”, que vai resultar na ação programada para aquela tecla. Um clique do mouse em um botão ocasiona alguma reação do botão.



EVENTOS:

Vamos partir para um exemplo simples de um componente que integra um evento dentro de si mesmo. Podemos ver abaixo que não existem grandes diferenças em relação ao o que já foi visto. Aqui existe um botão com o parâmetro de OnClick que chama a função criada acima "MeuEvento", que tem a funcionalidade de printar no console do navegador a frase determinada. É interessante notar que nesse mesmo componente é passado um props chamado de "numero", conceito já visto anteriormente na apresentação, assim, logo se deduz que é possível criar o mesmo componente com diferentes props que executam um mesmo tipo de evento.

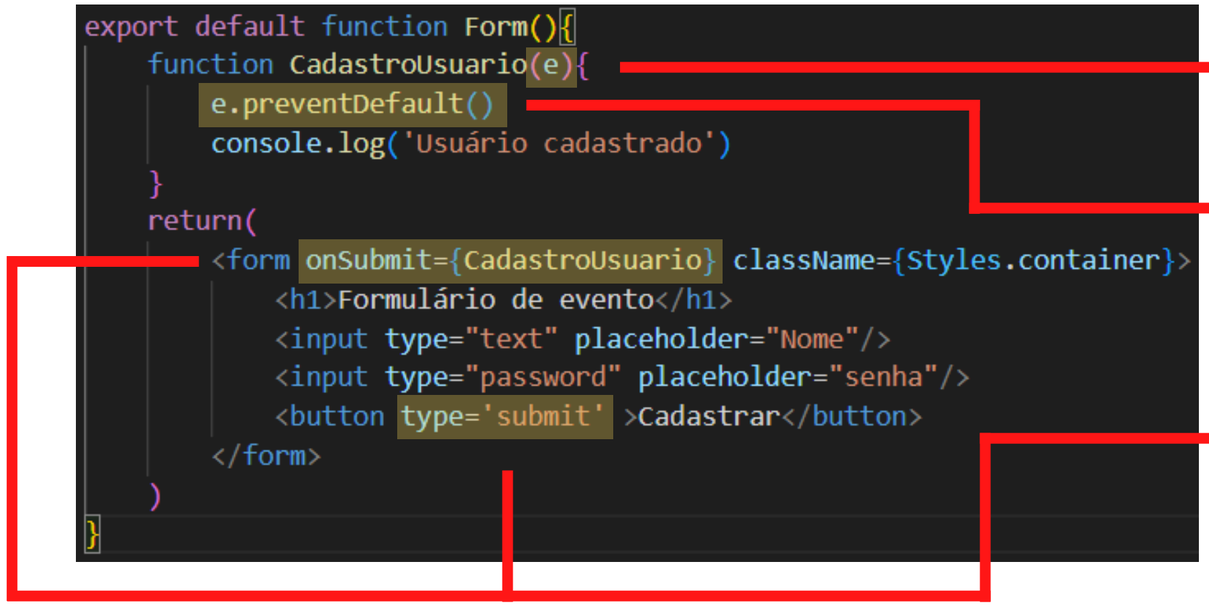


Utilização da props para poder receber o parâmetro "numero".

Evento disparado pelo Onclick do botão.

EVENTOS:

Aqui existe outro exemplo da utilização do conceito de evento, porém, agora com o formulário, geralmente os formulários são usados para poder requisitar informações do usuário da página web. O importante é observar que o evento vai ser disparado chamando a função "CadastroUsuario" no momento em que o formulário for submetido.



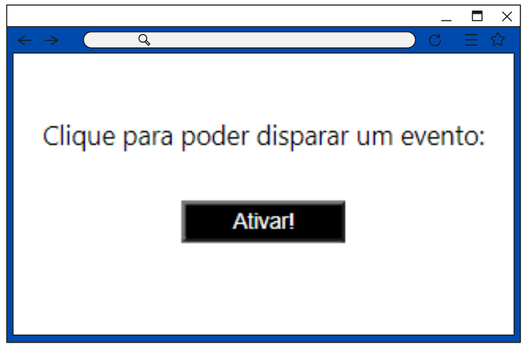
Evento sendo passado como "parâmetro" para a função.

O preventDefault tem a funcionalidade de evitar o reloading da página uma vez que estamos fazendo o envio de um formulário. Assim permitindo que a função seja executada com êxito.

Após o envio do formulário temos a chamada da função "CadastroUsuario" por meio do "onSubmit" que é ativado pelo botão com type "submit".

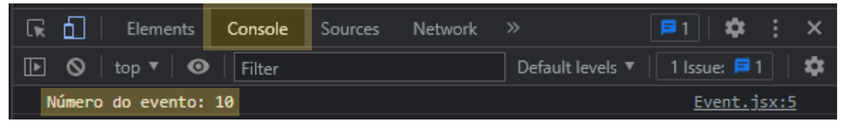
EVENTOS:

Após a criação desses componentes com eventos faltou apenas observar a integração e o resultado dos mesmos no navegador. Observe abaixo:

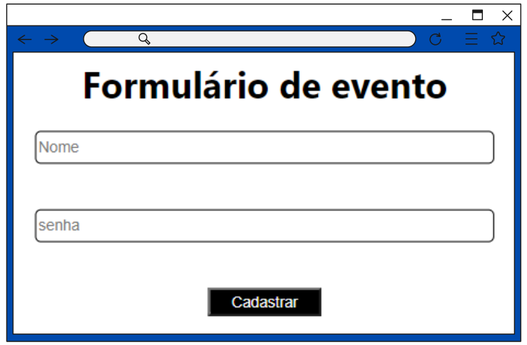


Componente Event:
Após apertar o botão "Ativar!" o evento é disparado com "OnClick" executando a função e printando no console do navegador a frase "Número do evento: 10", lembrando que o "10" se trata de uma props passada para o componente e utilizado na função.

Resultado no console:

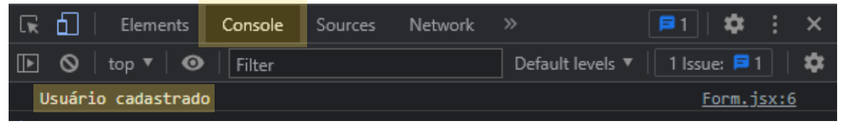


Número do evento: 10



Componente Formulário:
No formulário após apertar o botão "Cadastrar" o evento é disparado por meio do "OnSubmit" executando a função que printa no console a mensagem "Usuário cadastrado".

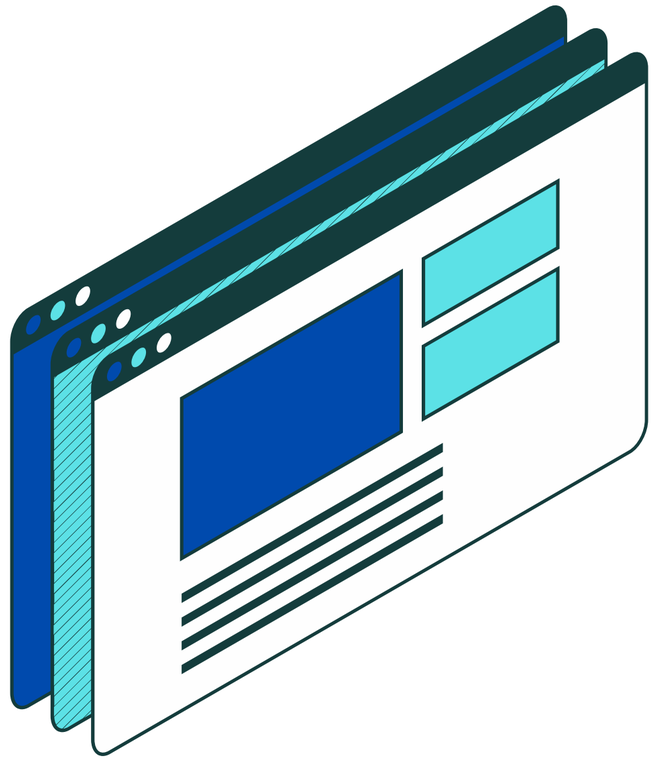
Resultado no console:



Usuário cadastrado

USESTATE:

- O useState é um **hook do React**;
- Com ele conseguimos **manusear o estado** de um componente de forma simples;
- Este hook funciona muito bem com eventos;
- Podemos **atrelar um evento** a mudança de state;



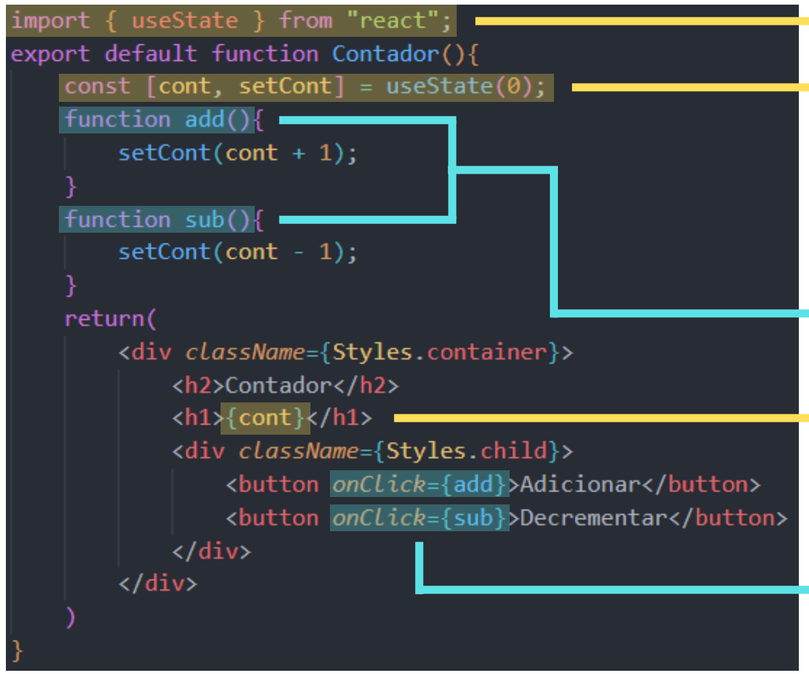
USESTATE:

O useState se trata de uma hook do React. Os hooks tem o objetivo de trazer para as funções funcionalidades que antes eram permitidas apenas via classe para os projetos, assim ajudando a organizar a lógica utilizada dentro dos componentes.

O useState é utilizado para poder atualizar o estado de um determinado dado, ao realizar uma mudança nesse dado precisamos atualizar o mesmo tanto no programa quanto na página web, assim, para que ambos estejam alinhados e correspondam ao que é esperado.

USESTATE:

Para poder exemplificar o funcionamento do useState vamos criar um componente chamado de "Contador" dentro do nosso projeto. A função desse componente vai ser de adicionar um contador funcional.



```
import { useState } from "react";
export default function Contador(){
    const [cont, setCont] = useState(0);
    function add(){
        setCont(cont + 1);
    }
    function sub(){
        setCont(cont - 1);
    }
    return(
        <div className={Styles.container}>
            <h2>Contador</h2>
            <h1>{cont}</h1>
            <div className={Styles.child}>
                <button onClick={add}>Adicionar</button>
                <button onClick={sub}>Decrementar</button>
            </div>
        </div>
    )
}
```

Importação do useState da biblioteca React.

Declaração da variável "cont" usando o useState. Observe que o declaramos como se fosse um array com dois índices. O primeiro valor deste array é uma variável que guarda o estado em si. Já o segundo valor é uma variável que é uma função, e é através dela que faremos as atualizações do valor do nosso estado.

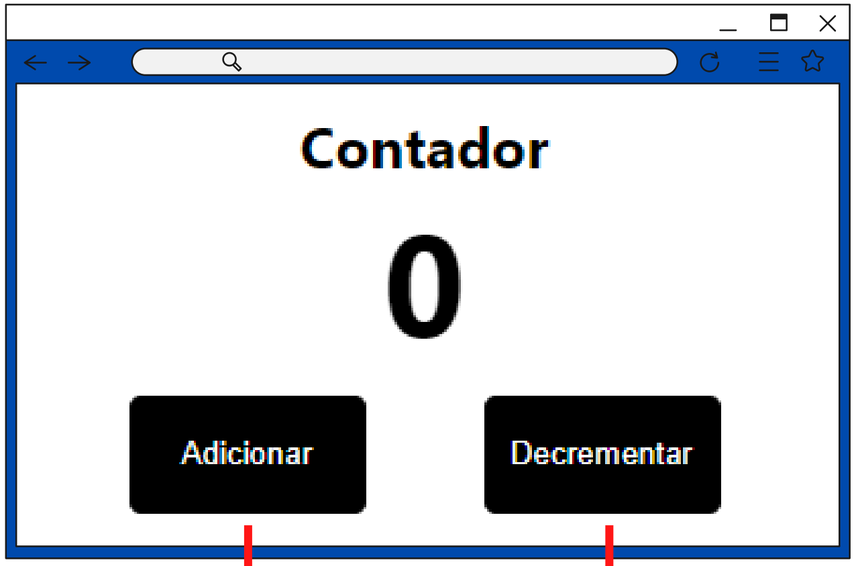
Aqui temos duas funções, o "add" e o "sub", ambos tem o objetivo de atualizar o estado do "cont", porém, uma subtrai o seu valor e a outra acrescenta por meio do "setCont".

Utilização da variável "cont", observe que a mesma está envolvida por chaves.

Por meio do OnClick de cada botão temos a chamada das funções que vão ficar responsáveis por acrescentar e decrementar o valor da variável "cont". Assim fazendo o contador atualizar.

USESTATE:

Visualize abaixo como o componente Contador se apresenta no navegador. Ao clicarmos no botão "Adicionar" o número que se apresenta a principio como 0 soma mais 1, e quando clicamos em "Decrementar" ele subtrai 1.



Dispara o evento que aciona a função "add".

Dispara o evento que aciona a função "sub".

RENDERIZAÇÃO CONDICIONAL:

- Podemos atrelar a exibição de algum elemento a um **if**;
- Esta ação é chamada de **renderização condicional**;
- Envolvemos as tags em **chaves {}**;
- Como as chaves **executam JavaScript,** criamos nossa condição;
- É possível **usar o state** para criar as condições;

RENDERIZAÇÃO CONDICIONAL:

O If é utilizado quando precisamos filtrar uma determinada informação, quando queremos que o dado atenda a um determinado requisito. Assim podemos colocar uma lógica de condição no front-end.

RENDERIZAÇÃO CONDICIONAL:

```jsx
export default function Condicional(){
    const [email, setEmail] = useState();
    const [userEmail, setUserEmail] = useState();

    function enviarEmail(e){
        e.preventDefault();
        setUserEmail(email);
    }
    function limparEmail(){
        setUserEmail('');
    }

    return(
        <form onSubmit={enviarEmail} className={Styles.container}>
            <h1>Cadastro de Email</h1>
            <input type="email" placeholder="Email"
                onChange={(e) => setEmail(e.target.value)}
            />
            <button type='submit' >Cadastrar</button>
            {userEmail && (
                <div className={Styles.child}>
                    <p>O e-mail do usuário é: {userEmail}</p>
                    <button onClick={limparEmail}>Remover E-mail</button>
                </div>
            )}
        </form>
    )
}
```

Descrição do programa:

Para poder desenvolver o programa exemplo utilizamos de vários conceitos vistos nos slides anteriores, um deles é o useState que tem o objetivo de definir o e-mail do usuário que está se cadastrando, porém, observe que temos duas variáveis, o "email" e o "userEmail", uma tem o seu valor definido no momento em que o usuário está digitando no input através do "onChange", enquanto a outra vai ser definida apenas quando o botão submit for pressionado chamando a função enviarEmail, na onde o valor da variável "email" vai ser atribuído ao "UserEmail".

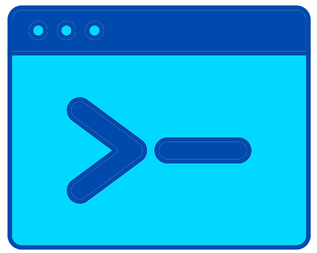
Isso foi feito para poder se manter uma ordem dentro da lógica, uma vez que temos uma renderização condicionar envolvida por um if no programa, if esse que está conferindo se a variável "userEmail" está atribuída com algum valor dentro de si, e como queremos que essa condição seja verdadeira apenas após o preenchimento do usuário e a confirmação de envio, utilizamos de duas variáveis.

Aqui temos acondição if que vai permitir a renderização da tag div apenas se "userEmail" tiver um valor atribuído em si mesmo. Fique atento ao fato de que a condição é envolvida por chaves {} o que permite atribuir lógica javaScript no meio do return do componente, outra fato é que o que vai ser renderizado deve estar completamente envolvido por uma única tag.

RENDERIZAÇÃO CONDICIONAL:

Aqui temos o que vai aparecer no navegador. Clicando no botão Cadastrar após preencher o campo input do Email teremos a renderização de uma div com o email que acabou de ser cadastrado e o botão "Remover E-mail", que tem a funcionalidade de "apagar" essa mesma div.

RENDERIZAÇÃO POR LISTA:



- Para poder renderizar uma lista é preciso ter um **array**;
- É utilizado a **função map**, para poder percorrer cada item da lista;
- É possível **unir operadores condicionais** com a renderização de listas;

RENDERIZAÇÃO POR LISTA:

```
const materias = [
  {
    nome : 'java',
    periodo : 'manha',
    professor : 'jorge',
    aulas : 30
  },
  {
    nome : 'python',
    periodo : 'tarde',
    professor : 'renato',
    aulas : 20
  },
  {
    nome : 'react',
    periodo : 'noite',
    professor : 'joao',
    aulas : 45
  }
]
```

Para poder realizar esse tipo de renderização primeiramente precisamos de uma lista, no caso ao lado temos uma lista chamada "matérias", essa lista tem diversos objetos com informações de matérias, como nome, período, professor e aulas. Ou seja, o objetivo vai ser percorrer uma lista de objetos e agrupar os seus dados de forma lógica e coerente. Essa lista foi criada no arquivo "App" e passada por meio de props para o componente "RenderizacaoLista" para poder ser percorrida, como podemos ver abaixo.

```
<RenderizacaoLista
  informacoes = {materias}
/>
```

RENDERIZAÇÃO POR LISTA:

Entrando no componente "RenderizacaoLista" passamos a ver a lógica por trás da renderização da lista de objetos. Para poder entender essa lógica vamos primeiramente separá-la em partes.

```
export default function RenderizacaoLista(props){
    return(
        <div className={Styles.container}>
            <h1>Renderização de lista</h1>
            {
                props.informacoes.length > 0 ? (
                    props.informacoes.map((item, index) => (
                        <div key={index} className={Styles.child}>
                            <h2>{item.nome}</h2>
                            <ul>
                                <li>Aulas: {item.aulas}</li>
                                <li>Periodo: {item.periodo}</li>
                                <li>Professor: {item.professor}</li>
                            </ul>
                        </div>
                    ))
                ) : (
                    <h2>Não existem matérias cadastradas</h2>
                )
            }
        </div>
    )
}
```

Primeiro devemos entender o "array.map()".

```
props.informacoes.map((item, index) => (
    <div key={index} className={Styles.child}>
        <h2>{item.nome}</h2>
        <ul>
            <li>Aulas: {item.aulas}</li>
            <li>Periodo: {item.periodo}</li>
            <li>Professor: {item.professor}</li>
        </ul>
    </div>
))
```

RENDERIZAÇÃO POR LISTA:

É importante sempre manter em mente que como essa se trata de uma lógica javaScript a mesma está completamente envolvida por chaves {} por mais que não apareça na print do exemplo abaixo.

A função map dentro do javaScript tem o objetivo de percorrer cada um dos itens dentro da lista "props.informacoes", como se fosse um "for" que varre cada posição.

```
props.informacoes.map((item, index) => (
    <div key={index} className={Styles.child}>
        <h2>{item.nome}</h2>
        <ul>
            <li>Aulas: {item.aulas}</li>
            <li>Periodo: {item.periodo}</li>
            <li>Professor: {item.professor}</li>
        </ul>
    </div>
))
```

O "item" se trata do objeto que está sendo varrido no momento, ele vai se alterando de acordo com que o map vai percorrendo a lista materias.

O index se trata de outro parâmetro que conseguimos resgatar do map, ele nos fornece a posição de cada item que está sendo percorrido dentro da lista, no programa ele está sendo utilizado para poder gerar uma key para a div que engloba as informações do objeto, fazemos isso para que o react possa se orientar e diferenciar um do outro, desta forma evitando avisos de conflito no console do navegador.

As partes do programa que estão **grifadas em verde** fazem referência as informações do objeto que está passando pelo "item" e veio da lista passada via props, como o nome, aulas, período e professor.

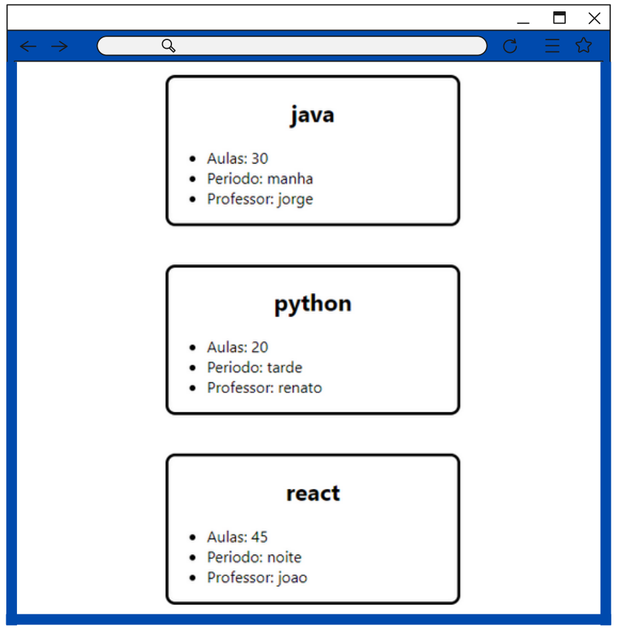
RENDERIZAÇÃO POR LISTA:

```
{
    props.informacoes.length > 0 ? (
        props.informacoes.map((item, index) => (
            <div key={index} className={Styles.child}>
                <h2>{item.nome}</h2>
                <ul>
                    <li>Aulas: {item.aulas}</li>
                    <li>Periodo: {item.periodo}</li>
                    <li>Professor: {item.professor}</li>
                </ul>
            </div>
        ))
    ) : (
        <h2>Não existem matérias cadastradas</h2>
    )
}
```

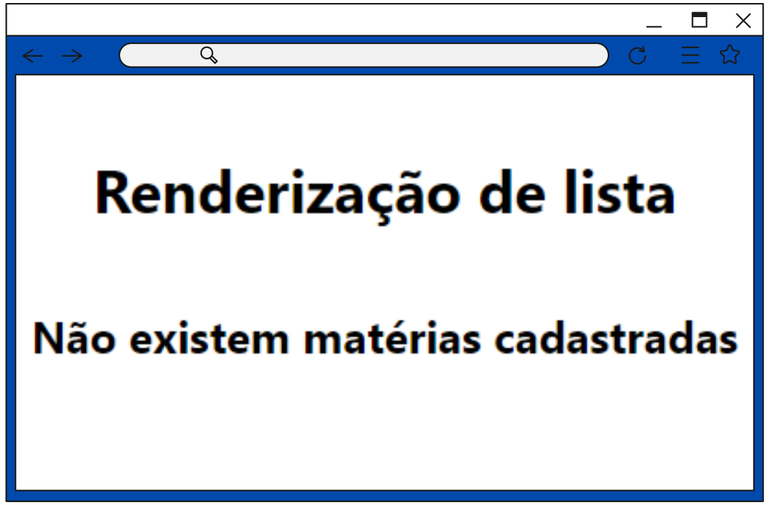
Agora englobando todo a parte do "map" temos um "if" que está conferindo o tamanho da lista, caso a lista seja maior que zero, logo significa que a mesma tem objetos para serem renderizados no navegador, caso o mesmo não atenda a condição ele acaba entrando direto no "else" representado pelo ":", assim deixando de renderizar a "div" para poder renderizar o "h2" com a mensagem "não existem matérias cadastradas" uma vez que não existem objetos dentro da lista.
Preste atenção nas chaves {} que estão grifadas em verde, elas estão ali para poder englobar toda a lógica javaScript que existe dentro do return do nosso componente, como já dito anteriormente nos slides.

RENDERIZAÇÃO POR LISTA:

Como resultado no nosso navegador vamos ter dois possíveis resultados dependendo se a condição do "if" vista no slide anterior for atendida ou não, caso seja, teremos a renderização das divs com as informações de acordo com o que estava dentro dos objetos, e caso não seja, teremos a renderização da tag "h2" informando que não existem matérias cadastradas.



Caso a lista seja esteja vazia este vai ser o resultado no navegador.



Me siga nas redes sociais:

Espero que tenha gostado!
Aproveite e me siga nas redes sociais :)

joao-montanari

João V. Montanari